Symbolic Model Interface

The Symbolic Model Interface (**SMI**) is a library which provides for the efficient construction and manipulation of symbolic representations for finite state systems, in particular for communication protocols. It is developped at the Verimag laboratory and is a part of the CADP toolset.

With **SMI**, a system is described as a network of communicating processes, each process being an extended finite state automaton. Given the system textual description, the **SMI** routines build an equivalent symbolic representation using decision diagrams. This representation can be the starting point in the development of more customized applications, e.g. model checking for different temporal logics or symbolic minimization with respect to equivalence relations. On this page you can find :

Principles

The input language

The primary purpose of the **SMI** input language is to allow a natural and concise description of processes and their communications. Secondly, this description must be simple enough to be used for the construction of a symbolic representation using decision diagrams(**DD**). Some of the input language concepts are sketched in the following.

A *process* encapsulates data and behavior. It is specified by its variables, its communication ports and one or more control threads.

Variables are used to store local information about the process. Currently, only variables with simple finite domains are allowed: booleans, bounded naturals and other finite sets. The scope of a variable is the process definition,i.e. variables cannot be shared between different processes.

A *thread* is defined as a *finite state automaton* whose transitions can test and assign process variables. Furthermore, a transition can contain a message *emission/reception*to/from one communication *port*. If a process contains more than one thread, they are completely asynchronous. The process executes a step by non-deterministically choosing a thread, then executing one of its enabled transitions. Note however that different threads can share the same process variables.

A *protocol* is specified by a *composition expression* whose ground terms are processes. Two operators are provided, one for *parallel composition with synchronization* by message emissions and receptions, and the other for *abstraction*of communication at some ports. The synchronization forces two or more transitions from different processes to be combined and executed simultaneously (*rendez-vous*). The abstraction hides or ignores the communication at some ports.

The symbolic model

The protocol *model* is a labeled transition system (Q, Act, $\{Ta \mid a$ in Act$\}$, Qo). $Q$is a finite set of states, Act is a set of actions, Ta subset of Q x Q is the transition relation labeled by $a$and Qo subset of Q is the set of initial states. Usually, such model underlies the protocol verification algorithms. Given a protocol described in our input language, our aim is to build and to handle efficiently its corresponding model.

The symbolic model interface was designed to allow the manipulation of model *state sets* and *transitions*, symbolically represented by decision diagrams.

The *symbolic model* is builded from a protocol described using the input language. More precisely, the protocol transition relations *Ta* are encoded using **DD**s. The encoding process can take into account a number of parameters. For example, the parallel composition and abstraction semantics can be considered CSP-like (with binary rendez-vous and CSP restriction) or CCS-like (with n-ary rendez-vous and CCS abstraction). We can also impose the computation of reachable states *Acc* - some verification algorithms are more efficient when the reachable states are a priori known.

Basic operations on sets, such as union, intersection or complementation are directly mapped to **DDs** functions. The inclusion or the equality test are straightforward using **DD**s. Some specialized functions which perform *model exploration*, e.g. to compute the initial state set Qo, or the successors/predecessors for a given state set, *Post*and *Pre* are also provided.

Finally, **SMI** is not based on one particular **DD** implementation, thus it can be used with any **DDs** and only a minimal interface is required.

Applications

- mu-calculus model checker
- minimal model generator

Mu-calculus model checker

The *model checker* performs the backward evaluation of alternating free mu-calculus formulae over symbolic model representations.

Intuitively, the semantics of a formula *P* represents the set of states which satisfy it and is noted by [[*P*]]. The model checking algorithm for a formula *P*o works in two steps:

- the set [[*P*o]] *is constructed* recursively over the formula structure.
- a *decision procedure*is invoked :
    - standard evaluation -:- checks if *Q*o is subset of [[*P*o]]
    - forward analysis -:- checks if *Acc* intersected with [[*P*o]] is not empty
    - invariant checking -:- checks if *Q*o is subset of [[*P*o]] and *Post(*[[*P*o]]) is subset of [[*P*o]]

Minimal model generator

Given a labeled transition system(LTS) the *minimal model generator*(**MMG**) generates an equivalent minimal LTS. The minimality is relative to a bisimulation equivalence. A precise and complete description of the **MMG** algorithm can be found in Bouajjani-Fernandez-Halbwachs-90-b.

Briefly, the principle of the **MMG** algorithm is to refine an initial partition of the state space until a *reachable* and *stable* partition is obtained. It can be also defined as a computation of the greatest fixed point of a *split*function defined over partitions. Different reductions (by strong, weak, branching,... bisimulation) are obtained considering an appropriate *split*function.

The algorithm can work with the symbolic model representations. More precisely, a symbolic representation of partitions can be used, i.e. representing each equivalence class by a decision diagram. All partition transformations are then reduced to classical operations on decision diagrams. Currentlly, an operational version of this algorithm works with the **SMI** library.

Related Papers

Examples

- Alternating Bit Protocol
- Fischer Mutual Exclusion Protocol

Alternating Bit Protocol

The alternating bit protocol is part of the 4th OSI transport layer. It allows the exchange of messages betweeen two entities, a transmitter and a receiver, linked by unreliable communication channels.

The protocol is composed by four asynchronous processes: a transmitter, a receiver and the two communication channels between them. The communication is performed via 6 ports, which link the processes and their environnment. The **SMI** description of this protocol is given in the following files:

bitalt.exp	transmitter.aut
bitalt.types	medium1.aut
bitalt.order	receiver.aut
	medium2.aut

Fischer Mutual Exclusion Protocol

This protocol can be easily described using multiple control threads and shared variables. Each process is defined as a control thread of one **SMI** master process. Another special thread is used to intialize the clocks Ci and to encode the discrete *time progression*. The complete description of the protocol with 2 processes is given in the following files.

fischer.exp	
fischer.types	fischer.aut
fischer.order	

Manual Pages

- Description
- Synopsis
- Extended Automata

- Networks of Extended Automata
- Data Types
- Variable Ordering

Description

The **SMI** library provides an interface to access finite models builded from *Networks* of *Extended Automata*. The models are represented symbolically using *Decision Diagrams*(**DDs**). The **SMI** implements functions to handle model state sets and model transitions. The **SMI** library use exactly one of the following **DD** implementation at time:

- *BMDD*s - Binary Multivalued Decision Diagrams (local)
- *Cudd*s - Colorado University Decision Diagram
- *TiGeR* - TiGeR Binary Decision Diagrams
- *Bdd*s - Binary Decision Diagrams (local)

We have been developped two applications using the **SMI** libraries: **evaluator** which evaluate mu-calculus formulae on the model and **mmg** which generate the minimal model w.r.t. some bisimulations. The **SMI** libraries and the applications are available for SunOS and HP-UX platforms.

Synopsis

- **evaluator** [smi-options] [eval-option] *name*[.exp] [*name2*]
- **mmg** [smi-options] [mmg-options] *name*[.exp]

The *name* denotes the use of the followings files to build the symbolic model representation:

- *name.**exp***-:- model definition
- *name.**types*** -:- model types
- *name.**order***-:- model variables order

The *name2* denote an optionally mu-calculus formula file.

The **smi-options** available are: **-stat** print various statistics during computation (not a default option) **-mem** *n* allocate *n* MB of memory for **DDs** (default $n = 8$) **-vars** *v* use at maximum *v* **DD** variables (default $v = 48$) **-sift** enable the DD variables *automatic reordering* (not a default option) **-sim** use the *simultaneous composition* of automata (not a default option) **-front** *frontier strategy* for the computation of reachable states (not a default option) **-noreach** not compute the reachable states (not a default option)

The **eval-options** might be one of the following: **-eval** formula *backward evaluation* (default option) **-path** extract an *execution path* to a satisfying state (not a default option) **-inv** simple *invariant checking*

(not a default option)

For the **mmg-options** see Aldebaran manual page.

Extended Automata

The extended automata are described using the following context-free grammar :

	Extended automata
Ext-Aut	::=Var-Decl-List Ctrl-Thread-List
Var-Decl-List	::=Var-Decl-List Var-Decl
	| None
Ctrl-Thread-List	::=Ctrl-Thread-List Ctrl-Thread
	| None
	Variable declarations
Var-Decl	::=Var-List ':' Type-Name
Var-List	::=Var-List ',' Var-Name
	| Var-Name
	Control threads
Ctrl-Thread	::=Thread-Descr Thread-Trans-List
Thread-Descr	::='**des**' '(' Init-State ',' Trans-Number ','States-Number ')'
Thread-Trans-List	::=Thread-Trans-List Thread-Trans
	| None
	Thread transitions
Thread-Trans	::='(' Start-State ',' Guard Action Assign ',' Final-State ')'
Guard	::='[' Bool-Expr ']'
	| None
Action	::='**send**' Gate-Name Out-Expr-List
	| '**receive**' Gate-Name In-Var-List
	| '**i**'
	| None
Out-Expr-List	::=Out-Expr-List '!' Expr
	| None
In-Var-List	::=In-Var-List '?' Var-Name
	| None
Assign	::=Var-Name ':=' Expr
	| Assign Assign
	| Assign ';' Assign
	| '{' Assign '}'
	| None
	Expressions
Expr	::=Bool-Expr
	| Nat-Expr
	| Enum-Expr
Bool-Expr	::=Var-Name
	| '~' Bool-Expr

```
                 | Bool-Expr Bool-Op Bool-Expr
                 | Enum-Expr Rel-Op Enum-Expr
                 | Nat-Expr Rel-Op Nat-Expr
Nat-Expr         ::=Var-Name
                 | Nat-Number
                 | Nat-Expr Nat-Op Nat-Expr
Enum-Expr        ::=Var-Name
                 | Enum-Item-Name
                 | Enum-Op Enum-Expr
                   Operators
Bool-Op          ::='V'
                 | '/\'
                 | '=>'
                 | '<=>'
Rel-Op           ::='='
                 | '<='
                 | '>='
                 | '>'
                 | '<'
                 | '<>'
Nat-Op           ::='+'
                 | '-'
                 | '*'
                 | '/'
                 | '%'
Enum-Op          ::='succ'
                 | 'pred'
```

Data Types

```
Type-Defs        ::=Nat-Def Enum-Def-List
                    Naturals range
Nat-Def          ::='nat' '{' Nat-Range '}'
                 |  None
Nat-Range        ::=Nat-Number
                    Enumerated types
Enum-Def-List    ::=Enum-Def-List Enum-Def
                 |  None
Enum-Def         ::='enum' Enum-Name '{' Enum-Item-List '}'
Enum-Item-List   ::=Enum-Item-List ',' Enum-Item-Name
                 |  Enum-Item-Name ',' Enum-Item-Name
Enum-Name        ::=Identifier
Enum-Item-Name::=Identifier
```

Variable Ordering

The variables and thread control states order used to build the symbolic model can be specified using an external file, which must respect the following syntax :

```
Order           ::=Order-Item-List
Order-Item-List::=Order-Item-List  Order-Item
                | None
Order-Item      ::=Aut-Name ':' Var-Name
                |  Aut-Name ':' Thread-Id
Aut-Name        ::=Identifier
Var-Name        ::=Identifier
Thread-Id       ::=Nat-Number
```

Download

The SMI toolset can be downloaded here.

Author : Marius Dorel Bozga

Contact the Webmaster.